A Precise Way to Propagate JML Annotations for Security Automata

Danny Bergsma
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
d.bergsma@student.utwente.nl

ABSTRACT

As security is an important concern in many areas, often security policies are defined that applications in these areas should obey. These policies can be conveniently formalized as security automata, which can be used to monitor applications at run time. However, this kind of validation is not always feasible or desirable, as it only reveals violations when they are already about to occur.
Static verification is a way to check adherence at compile time. This requires Hoare logic style annotations to be provided, which is error-prone and time-consuming. Earlier work [6] provides a method that assists in this by translating security automata into JML annotations. However, this method only produces annotations for the methods included in the automaton, so-called core annotations.
To be able to verify the correctness of the application's usage patterns of the core methods, the security policy should be *woven* throughout the application: previous work [9] presents a method that propagates the core annotations to the methods (indirectly) invoking the core-annotated ones. This method, however, has severe limitations. We have developed a very precise algorithm that overcomes the most important limitations: Where the old algorithm cannot propagate the kind of annotations the translation produces, ours *can*, even when they are getting complex. Our algorithm supports all types of variables and considers branching, as opposed to the old one, which can only propagate static specification-only variables.
We have applied our method to complex examples; where the old algorithm produced annotated Java code the static checker could not prove to be correct, our algorithm produced code the static checker *could* prove to be correct.

Keywords

security automata, JML, annotations, static verification

1. INTRODUCTION

In many areas, for example in those that deal with privacy-sensitive information, security is an important concern. Therefore security policies are often defined for applications in these areas. These policies can be conveniently formalized as security automata. Security automata can for example be used to limit the number of calls to a specific method, e.g. to limit the number of SMS messages that can be sent. Security automata can also specify that some method should not be called from within some other method. An example is given in Section 2.2.

By using the corresponding automata to monitor applications at run time, their adherence to the policies can be validated [10]: When the monitored application enters or exits a security-related method, the automaton updates its internal state. If the automaton reaches an illegal state, it will report a security violation. However, such a dynamic verification is not always desirable: usually one wants to know *a priori*, not afterwards, whether an application may violate a given security policy. Another reason is that the processing power of the platform on which the application runs, is too limited to make monitoring feasible.

Therefore often other verification methods are to be used. One approach is to perform a manual code review, but this is time-consuming and error-prone. Another way is to verify adherence by static checking: just based on the source code of the application. To make this work, Hoare logic style annotations should be provided to the static checker. These annotations specify the security policies that should be adhered to by the application. This means a lot of work; not only the methods directly related to security need to be annotated, but also the methods that (indirectly) invoke these ones. Also, the security experts may fully understand the details of the security automata they define, but not those of the specification language.

Algorithms and tools have been developed to assist in providing the required annotations. Pavlova *et al.* [9] propose a method in which the user only needs to annotate the methods directly related to the security policies. These so-called core annotations are algorithmically propagated to all methods (indirectly) invoking the directly related ones. Huisman and Tamalet [6] extend this method by providing a translation from security automata to core annotations.

However, both methods have limitations: The translation yields a type of core annotations, which we call *inlined* core annotations, that the propagation algorithm cannot use as input. The algorithm can only propagate so-called static ghost variables, a kind of specification variables; program (static, instance, local and parameter) and instance ghost variables are not supported. Also, the propagation algorithm does not consider branching. Consequently, the checker used for verification may produce many false alarms; as some proof obligations may be too strong, it is possible that some of them cannot be discharged.

We have developed a very precise algorithm that overcomes these limitations: First, it converts the inlined core annotations to pre- and postconditions of new methods, which our propagation algorithm can use as input, as it propagates pre- and postconditions more precisely; the old algorithm would still be too limited to propagate them cor-

17th Twente Student Conference on IT June 25st, 2012, Enschede, The Netherlands.

rectly for all but the most simple cases. Our algorithm also takes care of pre- and postconditions involving method arguments and results. This is an example of our support of other types of variables (program and instance ghost variables). Finally, our algorithm now considers branching.

As *propagating* annotations can be considered as a special case of *generating* annotations, we investigated how techniques for generating other kinds of annotations could be used in our algorithm, e.g. the generation of loop invariants. Strongest postcondition calculus and, related, symbolic execution [8], are such techniques. Because of the strong resemblance with propagating preconditions, we also used elements from weakest precondition (wp) calculus [7] for the development of our algorithm.

We have applied the old and our algorithm on some complex examples that involve security automata. The input was Java code with inlined core annotations, the output fully annotated Java code: the inlined core annotations converted to pre- and postconditions of new methods, propagated to all methods (indirectly) invoking the core-annotated ones. We have given the fully annotated code as input to a static checker, which was able to prove that the annotated code produced by our algorithm was correct, as opposed to the code produced by the old one.

Pavlova *et al.* [9] used JACK [3] as static checker, but this tool is no longer maintained. We chose ESC/Java2 [4], as it is a complete tool and we have experience with it.

This paper is organised as follows. Section 2 introduces the background concepts of JML and property automata. Next, Section 3 summarizes the previous work on propagating annotations for security automata and discusses its limitations. Section 4 presents an overview of our solution. Section 5 presents our conversion from inlined core annotations to pre- and postconditions of new methods. Next, Section 6 presents our algorithm for propagating the just generated annotations. Section 7 summarizes the steps that are to be taken to go from security automata to fully annotated code. Section 8 presents related work. We conclude in Section 9, where we draw conclusions and discuss future work. The detailed and formal version of our algorithm is included in the appendices; a version with more explanation is available online: `http://goo.gl/bU1v9`

2. BACKGROUND

This section introduces three concepts that are used in the remainder of this paper: JML, in particular static (class) ghost variables, and property automata.

2.1 JML

The Java Modeling Language (JML) [2] is considered as the most important behavioural specification language for Java programs. By providing JML annotations, programmers specify properties of their programs (predicates) in a generalisation of Hoare logic, adapted to Java. JML's syntax is reminiscent of Java's, but it adds some specific keywords, such as `\forall`. Method pre- and postconditions are specified by `requires` and `ensures` clauses, respectively. So-called `assignable` clauses specify what variables the specified method may modify. Invariants, which need to be preserved by all methods, are specified by the keyword `invariant`. Tool support is available for verifying the provided annotations.

JML supports specification-only variables, so-called ghost variables, as a means of abstraction. They can be declared as instance or statically, and can be used in `requires`, `ensures` and `assignable` clauses. They are assigned to by a `set` annotation. An example is given in Section 3.

2.2 Property automata

Figure 1, adapted from [6], shows an example of a security automaton. It is meant to limit the number of calls to the method `sendSMS`: This method can be called successfully at most `N` times in between calls to the method `reset`. The counter `n` is only increased on a successful exit from `sendSMS`. The method `reset` resets the counter `n` to zero; it should not be called from within `sendSMS`.

The security automaton from Figure 1 is an example of a so-called property automaton (PA), an extended finite state machine (FSM). It consists of control points (s_1 and s_2), automaton variables (n), program variables (none in the example) and transitions. The current control point and the valuation of all automaton variables define the current automaton state. PA's transitions do not only depend on this state, but also on the program state. Transitions change the automaton state from one to another. For example, the upper transition changes the automaton state from (s_1, n) to (s_2, n), but only if $n < N$ upon entry of the method `sendSMS`; to indicate a security violation, no transitions can be applied under the condition $n \geq N$. A security automaton thus specifies all legal traces.

More formally, a transition consists of a guard, an event and a list of actions. The event specifies the method whose entry or (exceptional) exit is being monitored. The guard specifies the condition under which the transition can be applied. This condition refers to the automaton state, the program state, and method arguments (entry), result (normal exit) or exception (abnormal exit). Actions describe how the transition updates the automaton state. In the upper transition example, `entry(sendSMS)` is the event, `n < N` the guard and ϵ the (empty) action. [6]

3. PREVIOUS WORK

As mentioned above, Pavlova *et al.* [9] present an algorithm to propagate core JML annotations to the methods invoking the core-annotated methods. They suggest several core annotations for Java Card applets. Java Card is a dialect of Java, tailored to smart card applets. Java Card supports a transaction mechanism to provide atomicity: if something goes wrong during the transaction (e.g. the smart card is brutally retrieved from the terminal), the card will return to the state before the (not completed) transaction. Transactions are initiated by `beginTransaction()` and committed by `commitTransaction()`.

Several security policies related to the transaction mechanism can be formulated, such as 'no nested transactions'. The corresponding JML core annotations are synthesized as follows: First, we declare a static ghost variable `TRANS` and initialize it to zero:

```
@ static ghost int TRANS = 0
```

This variable keeps track of the number of ongoing transactions; initially, this number is zero. The next step is to annotate the `beginTransaction()` method:

```
@ requires TRANS == 0
@ assignable TRANS
@ ensures TRANS == 1
```

This annotation states that the method should only be called when there are no ongoing transactions (precondition $TRANS = 0$), that the `TRANS` variable may be changed by the method and that there is exactly one ongoing transaction when the method has returned (postcondition $TRANS = 1$). The annotation `@ set TRANS = 1` should be added to the body, as otherwise the method would not satisfy its specification. Similar annotations can be synthesized for `commitTransaction()`.

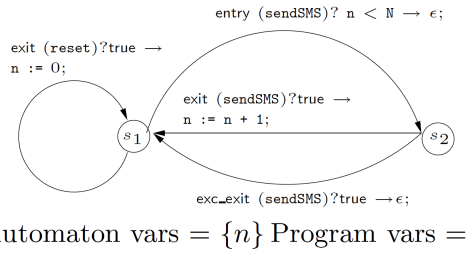

Figure 1. Property automaton example

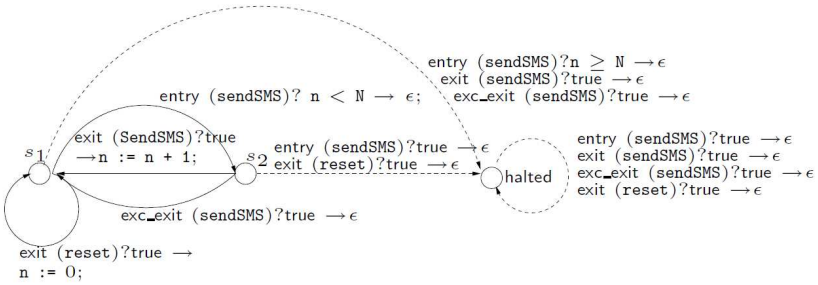

Figure 2. Automaton of Figure 1, after completion

Providing only core annotations is not enough for static verification. Consider method `m()`:

Code Snippet 1 Body `m()`

```
beginTransaction();
commitTransaction();
```

The static checker will report an error, as it cannot guarantee that `m()` will satisfy the precondition of `beginTransaction()` ($TRANS = 0$) at the time of calling it; the precondition $TRANS = 0$ should be propagated to the specification of `m()`. Now the static checker *can* guarantee that the precondition of `beginTransaction()` will be satisfied at the time of calling it. As `beginTransaction()` may modify `TRANS`, any precondition of `commitTransaction()` that involves $TRANS$ is *not* propagated. Conversely, any postcondition of `commitTransaction()` should be propagated; the postcondition of `beginTransaction()` ($TRANS = 1$) is not propagated, as `commitTransaction()` may modify `TRANS`. Pavlova *et al.* [9] present such a propagation algorithm, that takes into account this kind of considerations. The algorithm thus *weaves* the security policy throughout the application.

From property automaton to core annotations.

The core annotations still have to be provided by the user, which can be hard and time-consuming for complex security policies. To overcome this limitation, Huisman and Tamalet [6] present a translation from property automata to core annotations. Essentially, they consider security automata as specifications.

First, they translate any partial automaton to its total one. A PA is partial if for some source control point (cp) and event there is *not* always a guard that holds. The automaton in Figure 1 is partial: for example, if the automaton is at cp s_1, the event `entry(sendSMS)` occurs *and* $n \geq N$, no guard holds. A total automaton introduces the new cp `halted` and adds transitions to this cp for all "missing" combinations of source cps, events and guards, which should be halted, as they indicate a security violation (illegal trace). See Figure 2, again adapted from [6], for the result of translating the partial automaton of Figure 1 to its total one.

Next, they generate annotations for the methods included in the total automaton. These capture the behaviour of the automaton and are so-called method-level CaseSet annotations that are "evaluated" upon entry and exit of a method. Finally, they inline these CaseSets into the method bodies. For example, the method `sendSMS` from the total automaton of Figure 2 is converted to[1]:

[1]Our `CaseSet` syntax is different from the one in [6]: our syntax eliminates the nested `CaseSets`. The semantics, however, remain the same.

Code Snippet 2 `sendSMS`: inlined CaseSets

```
@ CaseSet[(cp == s1 && n < N, cp = s2),
@         (cp == s1 && n >= N, cp = halted),
@         (cp == s2, cp = halted),
@         (cp == halted, cp = halted)]
// body
@ CaseSet[(cp == s1, cp = halted),
@         (cp == s2, cp = s1; n = n+1),
@         (cp == halted, cp = halted)]
```

A CaseSet element is a pair: the first part being the condition (current cp and (possibly the default) guard) and the second part the new cp and zero or more assignments to automaton variables. The semantics of the CaseSet are as follows. First, the condition from every CaseSet element is evaluated; exactly one condition evaluates to *true*. Next, the second part of this CaseSet element is executed, i.e. updating the cp and possibly automaton variables. The variables `cp` and `n`, and constants `N`, `s1`, `s2` and `halted`[2] are ghost ones. An invariant states that the cp should not become the halted one:

```
@ invariant cp != halted
```

3.1 Limitations

However, the propagation algorithm in [9] cannot use these inlined CaseSet annotations as input; it needs method pre- and postconditions as core annotations. Therefore we need to adapt the algorithm in such a way that it will accept the inlined CaseSet annotations as core annotation input.

The propagation algorithm has other limitations as well: It propagates only static ghost variables; instance ghost and program (static, instance, local and parameter) variables are not (explicitly) supported. Also, it does not consider branching. Consider method `branch(boolean b)`:

Code Snippet 3 Body `branch(boolean b)`

```
if (b) beginTransaction();
else   commitTransaction();
```

The old algorithm just propagates the preconditions of `beginTransaction()` and `commitTransaction()`

$$(TRANS = 0 \wedge TRANS = 1) \equiv \mathit{false}$$

to the contract of `branch()`. This precondition is so strong that it cannot be satisfied. A precise and satisfiable precondition would be:

$$(b \wedge TRANS = 0) \vee (\neg b \wedge TRANS = 1)$$

Thus, the old algorithm generates too strong preconditions and other imprecise annotations. Consequently, the static checker may reject correct applications, as the too strong proof obligations cannot be discharged. The algorithm's imprecision, however, could not cause *incorrect* applications to be accepted by the checker.

[2]In [6] the constants are variables as well. ESC/Java2, however, can only handle them as constants, so we adapted the translation from automaton to annotations slightly.

4. APPROACH

As the propagation algorithm cannot use the inlined CaseSets as core annotation input, we first convert the CaseSets: each CaseSet is converted to a new method with appropriate pre- and postconditions; the CaseSet is replaced with a method call to this new method. See Section 5 for details. Then, we propagate the just constructed pre- and postconditions to the contract of the method for which the (meanwhile replaced) CaseSets were defined. The old propagation algorithm, however, would still be too limited to propagate the pre- and postconditions correctly for all but the most simple cases. Therefore, we have adapted the propagation algorithm in such a way that it correctly propagates them, even if they are complex. Additionally, we have added support for the other types of variables, such as method parameters, and support for branching. Our enhanced propagation algorithm is explained in Section 6.

5. CONVERSION ALGORITHM

The detailed conversion algorithm is given in Appendix B; in this section we explain the general idea.

Each CaseSet is converted to a new method with appropriate pre- and postconditions, `assignable` clause and body; the CaseSet is replaced with a call to this new method. The precondition consists of the disjunction of the conditions of all CaseSet elements that do *not* set the cp to halted. The `assignable` clause consists of `cp` and all automaton variables that are assignee in one or more CaseSet elements. The postcondition states for the condition of each CaseSet element that does not set the cp to halted, the new cp and the result of any assignments to automaton variables; if an automaton variable is not an assignee in a particular CaseSet element, but *is* an assignee in another CaseSet element, the postcondition for the former element additionally states that the automaton variable is not changed. An unreachable cp, say s, is captured in the additional postcondition $false \Rightarrow s$. The body consists of constructed `set` statements that change the cp and automaton variables according to the second part of all CaseSet elements.

For example, the first CaseSet of the method `sendSMS()` (see Code Snippet 2) is converted to the new method `_preSendSMS()` with Code Snippet 4 as JML method contract; the second CaseSet is converted to `_postSendSMS()`, see Code Snippet 5 for its contract and body.

Code Snippet 4 Contract `_presendSMS()`

```
@ requires cp==s1 && n<N
@ assignable cp
@ ensures (\old(cp)==s1 && n<N) ==> cp==s2
@ ensures false ==> cp==s1
```

Code Snippet 5 Contract and body `_postsendSMS()`

```
@ requires cp==s2
@ assignable cp, n
@ ensures \old(cp)==s2 ==> (cp==s1&&n==\old(n)+1)
@ ensures false ==> cp==s2
private void _postSendSMS() {
   @ set n = (cp==s2 ? (n+1) : n)
   @ set cp = (cp==s2 ? s1 : cp)
}
```

The CaseSets in the body of `sendSMS()` are replaced with method calls to `_preSendSMS()` and `_postSendSMS()`:

Code Snippet 6 Body `sendSMS()`

```
_preSendSMS(); //body _postSendSMS();
```

Note that in both contracts the antecedent in the first `ensures` clause is equal to the precondition, so the implication can be optimised away. This is, however, not possible if the precondition consists of multiple disjuncts: Suppose that we have an additional cp s_3, such that the first CaseSet (see Code Snippet 2) is extended with element

```
(cp==s3, cp = s3)
```

and the second CaseSet with element:

```
(cp==s3, cp = s1)
```

The contract of `_preSendSMS()` is now Code Snippet 7; the contract of `_postSendSMS()` Code Snippet 8.

Code Snippet 7 Contract `_presendSMS()` *(extended)*

```
@ requires (cp==s1 && n<N) || cp==s3          (...)
@ ensures (\old(cp)==s1 && n<N) ==> cp==s2
@ ensures \old(cp)==s3 ==> cp==s3             (...)
```

Code Snippet 8 Contract `_postsendSMS()` *(extended)*

```
@ requires cp==s2 || cp==s3                   (...)
@ ensures \old(cp)==s3 ==> (cp==s1 && n==\old(n))
@ ensures \old(cp)==s2 ==> (cp==s1&&n==\old(n)+1)
```

6. PROPAGATION ALGORITHM

The just constructed core pre- and postconditions need to be propagated, to the contract of `sendSMS()` in the first place. We explain our enhanced propagation algorithm in stages, adding complexity along the way. The detailed algorithm is given in Appendix C; in this section we give the general idea.

6.1 Sequential method calls

6.1.1 Propagating preconditions

The old algorithm does not propagate preconditions of method calls if they contain variables that are changed by an "earlier method call", e.g. in Code Snippet 1, the precondition of `commitTransaction()` is not propagated. It turned out that this approach was too limited for the purpose of our 'limited SMS' example. We adapted the precondition propagation algorithm in such a way that preconditions of method calls that contain variables that are changed by an "earlier method call" are kept, albeit adapted, when adapted preconditions can be derived from them by the postconditions of this "earlier method call". To this end, the method body is analysed in a sequential way from end to beginning (in contrast with the opposite sequential way, from beginning to end, of the old algorithm), propagating the preconditions of method calls; when our algorithm arrives at a method call, it determines if a precondition that is propagated and contains variables that are changed by this method, can be transformed to another condition by the method's postconditions. If this is the case, the precondition will *be* transformed and the adapted precondition will be kept, i.e. propagated.

We have identified three ways how postconditions can be used for the deriving of preconditions (assume a propagated precondition $a \wedge n < N$):

- A (sub)predicate of the propagated precondition is one of the conjuncts in the consequent of one or more implications, say I, in the postconditions; this part can then be replaced with the disjunction of all antecedents of I. For example, assume postconditions $\mathsf{old}(b) \Rightarrow a$ and $c \Rightarrow (a \wedge d)$. The precondition can then be transformed to $(b \vee c) \wedge n < N$.
- A part of the propagated precondition is one side of an equation in a postcondition; this part can then be replaced with the other side of the equation. For example, assume postcondition $n = \mathsf{old}(n) + 1$. The precondition can then be transformed to:

$$a \wedge (n + 1) < N$$

- A part of the propagated precondition is one side of an equation in an implication in the postconditions; this part can then be replaced with the other side of the equation, conditioned on the antecedent of the implication. For example, assume the postconditions $\mathsf{old}(g) \Rightarrow (n = \mathsf{old}(n) + 1)$ and $h \Rightarrow n = \mathsf{old}(n)$. The precondition can then be transformed to:
$$a \wedge (g \Rightarrow (n + 1) < N) \wedge (h \Rightarrow n < N)$$

Examples.

Recall the contracts of `_pre-` and `_postSendSMS()`, and the converted body of `sendSMS()` (Code Snippets 4-6).

The precondition of `sendSMS()` is calculated as follows. The precondition of `_postSendSMS()`, $\mathsf{cp} = s_2$, is transformed to $\mathsf{cp} = s_1 \wedge n < N$, which is equivalent to the integrally propagated precondition of `_preSendSMS()`, so it does not strengthen the precondition here; the old algorithm outputs the same precondition.

Now consider the contracts of the *extended* `_pre-` and `_postSendSMS()` (Code Snippets 7-8). The precondition of the extended `_postSendSMS()`, $\mathsf{cp} = s_2 \vee \mathsf{cp} = s_3$, is transformed to $(\mathsf{cp} = s_1 \wedge n < N) \vee \mathsf{cp} = s_3$, which is equivalent to the integrally propagated precondition of the extended `_preSendSMS()`, so again it does not strengthen the precondition here; again, the old algorithm outputs the same precondition.

The old algorithm, however, fails if the precondition of the extended `_postSendSMS()` (Code Snippet 8) is changed to $\mathsf{cp} = s_2 \vee \mathsf{cp} = s_1$: this precondition is transformed to $(\mathsf{cp} = s_1 \wedge n < N) \vee \mathit{false}$, which, in fact, strengthens the overall precondition of the extended `sendSMS()` to $\mathsf{cp} = s_1 \wedge n < N$ (removing disjunct $\mathsf{cp} = s_3$); the old algorithm just propagates the precondition of the extended `_preSendSMS()` (Code Snippet 7), which means that the disjunct $\mathsf{cp} = s_3$ is (incorrectly) kept.

6.1.2 Propagating postconditions

The old algorithm propagates postconditions in a similar, but converse, way as it propagates preconditions, e.g. in Code Snippet 1, the postcondition of `beginTransaction()` is not propagated. It again turned out that this approach was too limited for the purpose of our 'limited SMS' example. We adapted the postcondition propagation algorithm in such a way that postconditions, say P_1, of method calls that contain variables that are changed by a "later method call", are integrated into the postconditions, say P_2, of this "later method call": if a postcondition $p_1 \in P_1$ is an equation $a = b$, it is integrated into a postcondition $p_2 \in P_2$ by replacing in p_2 any pre-invocation value of either side of the equation ($\mathsf{old}(a)$ and $\mathsf{old}(b)$) by the other side of the equation (b and a, respectively). For example, if p_1 is $n = \mathsf{old}(n) + 1$ and p_2 is $n = \mathsf{old}(n) + 2$, $\mathsf{old}(n)$ in p_2 is replaced with $\mathsf{old}(n)+1$, resulting in $n = (\mathsf{old}(n)+1)+2$. Our *precondition* propagation algorithm analyses the method body *from end to beginning*; for similar reasons, our *postcondition* propagation algorithm analyses it *from beginning to end* (in contrast with the opposite way, from end to beginning, of the *old* postcondition propagation algorithm).

As postconditions of method calls may involve implications (i.e. are conditional), we use a variant of symbolic execution and keep track of *path conditions*. This condition is used to determine if the hitherto followed path implies the antecedent of a method call postcondition implication.

We first determine the possible precondition combinations of the method (body) for which we need to determine the appropriate postcondition; a possible precondition combination is the conjunction of an element from the Cartesian product of the disjuncts of all precondition conjuncts, e.g. if the precondition of a method is $(a \vee b) \wedge (c \vee d)$, its Cartesian product is $\{a, b\} \times \{c, d\} = \{[a,c], [a,d], [b,c], [b,d]\}$ and the conjunction of one of its elements is $a \wedge c$.

The method body is evaluated for each possible precondition combination, i.e. for each element from the Cartesian product, from beginning to end: First, the path condition is initialized with this combination. As the body is being evaluated, the path condition is updated with postconditions from method calls, but only if they do not involve an implication *or* if the path condition implies the antecedent of the implication. If a method call changes variables V, they are invalidated in the path condition by renaming them; any pre-invocation value $\{\mathsf{old}(v) \mid v \in V\}$ in the method call postconditions is replaced with the just renamed variable name. The ultimate goal of keeping track of the path condition is to determine which consequents from method call postcondition implications can be inferred and thus be propagated (and possibly integrated into (inferred) postconditions from later method calls). The resulting postcondition for the method body is an implication, with the original path condition, i.e. precondition combination, as antecedent and the postcondition "at the end of the method body" as consequent.

Examples.

Recall the contracts of `_pre-` and `_postSendSMS()`, and the converted body of `sendSMS()` (Code Snippets 4-6).

The postcondition of `sendSMS()` is calculated as follows. The only possible precondition combination is
$$\mathsf{cp} = s_1 \wedge n < N$$
and the path condition is initialized herewith. This condition implies the antecedent of the postcondition
$$(\mathsf{old}(\mathsf{cp}) = s_1 \wedge n < N) \Rightarrow \mathsf{cp} = s_2$$
of `_preSendSMS()`, so the path condition is updated to
$$cp_0 = s_1 \wedge n < N \wedge \mathsf{cp} = s_2$$
and the consequent $\mathsf{cp} = s_2$ is propagated. The updated condition implies the antecedent of the postcondition
$$\mathsf{old}(cp) = s_2 \Rightarrow (\mathsf{cp} = s_1 \wedge n = \mathsf{old}(n) + 1)$$
of `_postSendSMS()`, so the path condition is updated to
$$cp_1 = s_1 \wedge n_0 < N \wedge cp_0 = s_2 \wedge \mathsf{cp} = s_1 \wedge n = (n_0 + 1)$$
and the propagated postcondition $\mathsf{cp} = s_2$ is integrated into the consequent $\mathsf{cp} = s_1 \wedge n = \mathsf{old}(n) + 1$; as $\mathsf{old}(\mathsf{cp})$ does not occur in the consequent, the consequent is not changed and is integrally propagated. The resulting postcondition for `sendSMS()` is an implication with the original path condition, i.e. precondition combination,
$$\mathsf{cp} = s_1 \wedge n < N$$
as antecedent and the inferred postcondition
$$\mathsf{cp} = s_1 \wedge n = \mathsf{old}(n) + 1$$
as consequent. The full contract of `sendSMS()` is:

Code Snippet 9 Contract `sendSMS()`

```
@ requires (cp==s1 && n<N)
@ assignable cp, n
@ ensures (\old(cp)==s1 && \old(n)<N) ==>
@                          (cp==s1 && n==\old(n)+1)
```

The old propagation algorithm outputs
$$\mathsf{old}(\mathsf{cp}) = s_2 \Rightarrow (\mathsf{cp} = s_1 \wedge n = \mathsf{old}(n) + 1)$$
as postcondition, which is incorrect. Note, however, that if the implications in the contracts of `_pre-` and `_postSendSMS()` (Code Snippets 4-5) are optimised away, the old propagation algorithm will output a correct postcondition, as our enhanced algorithm will do. It is, however, not always possible to optimise away implications, see the con-

tracts of the *extended* `_pre-` and `_postSendSMS()` (Code Snippets 7-8). Our propagation algorithm outputs the correct contract for the *extended* `sendSMS()`:

Code Snippet 10 Contract `sendSMS()` *(extended)*

```
@ requires (cp==s1 && n<N) || (cp==s3)          (...)
@ ensures (\old(cp)==s1 && \old(n)<N) ==>
@                          (cp==s1 && n==\old(n)+1)
@ ensures \old(cp)==s3 ==> (cp==s1 && n==\old(n))
```

Note that our postcondition propagation algorithm evaluates the method body twice, as the precondition consists of two disjuncts. The old algorithm just propagates the postcondition of the extended `_postSendSMS()` (Code Snippet 8), which is not valid for the extended `sendSMS()`.

6.1.3 Complex example

Consider method `thrice()` which calls the extended `sendSMS()` (Code Snippet 10) thrice:

```
sendSMS(); sendSMS(); sendSMS();
```

Our propagation algorithm outputs an enormous contract for `thrice()`, with multiple `requires` clauses and multiple implications in them, and even bigger `ensures` clauses, each involving more than ten implications. However, the contract is equivalent to requiring

$$(\mathsf{cp} = s_1 \wedge (n+2) < N) \vee (\mathsf{cp} = s_3 \wedge (n+1) < N)$$

and ensuring

$$\mathsf{old}(\mathsf{cp} = s_1 \wedge (n+2) < N) \Rightarrow (\mathsf{cp} = s_1 \wedge n = \mathsf{old}(n) + 3)$$
$$\mathsf{old}(\mathsf{cp} = s_3 \wedge (n+1) < N) \Rightarrow (\mathsf{cp} = s_1 \wedge n = \mathsf{old}(n) + 2)$$

The contract can be used to prove, using ESC/Java2, that

```
@ assume (cp==s3 && n==3 && N==5)
thrice();
@ assert (cp==s1 && n==5)
```

and

```
@ assume (cp==s1 && n==2 && N==5)
thrice();
@ assert (cp==s1 && n==5)
```

are *correct*, and that

```
@ assume (cp==s1 && n==3 && N==5)
thrice();
@ assert (cp==s1 && n==6)
```

is *incorrect*. (The `assume` statement tells ESC/ Java2 that the given predicate is *assumed* to be *true*; the `assert` statement tells ESC/Java2 to *check* that the given predicate is *true* at the given point.)

The old propagation algorithm just outputs the same contract for `thrice()` as for `sendSMS()` (Code Snippet 10). As it does *not* incorporate $(n+2) < N$ (if $\mathsf{cp} = s_1$) and $(n+1) < N$ (if $\mathsf{cp} = s_3$) into its preconditions, and not $n = \mathsf{old}(n) + 3$ (if $\mathsf{old}(\mathsf{cp}) = s_1$) and $n = \mathsf{old}(n) + 2$ (if $\mathsf{old}(\mathsf{cp}) = s_3$) into its postconditions, this contract of `thrice()` can*not* be used to prove that the first two "assume/assert"s are correct.

6.2 More language constructs

So far, we have only defined rules for sequential method calls with no arguments or result values. In this subsection, we add rules for more constructs of Java: method arguments and results, assignments and branching. We also explain how we deal with local variables that may appear in the calculated postcondition of a method body.

6.2.1 Method arguments and results

Methods may have arguments, which may appear in their contracts. The same applies to the return values of methods. Consider the methods `y()` and `z()` (where `d` is a static ghost `int`)

Code Snippet 11 Signature of method `y()`

```
@ ensures \result==(a+2) && d==(a+3)
static int y(int a);
```

Code Snippet 12 Signature of method `z()`

```
@ requires b>5
@ ensures d==(\old(d)+b)
static void z(int b);
```

and method `x()`, which calls `y()` and `z()`:

Code Snippet 13 Method `x()`

```
static int x(int c) { z(y(c)); }
```

We want to propagate the pre- and postconditions of `y()` and `z()` to `x()`. As formal params have no meaning outside the method, we need to relate them to the actual params. E.g. in the method call `y(c)` (in `x()`) we relate the formal param a of `y()` to actual param (expression) c and we replace all occurrences of a with the postcondition of `y()` by c: $\mathsf{result} = (c+2) \wedge d = (c+3)$.

Note that method calls may appear in the actual params of *other* method calls, e.g. `y(c)` in `z(y(c))` (method `x()`). This makes that we need to propagate the pre- and postconditions of the former, e.g. the ones of `y(c)`. Also, as methods may exhibit side effects, we cannot relate formal params directly to method calls; instead, we try to relate it to any $\mathsf{result} = expr$ postcondition of the called method. For example, in the method call `z(y(c))` we relate the formal param b of the method `z()` to the $\mathsf{result} = c+2$ postcondition of actual param `y(c)`, which yields $c+2$ and not $a+2$, as formal param a is related to c in `y(c)` itself. As b is related to $c+2$, the precondition $(c+2) > 5$ is propagated to `x()`; the postcondition of `y(c)`, $d = (c+3)$, is integrated into the postcondition of `z(y(c))`, $d = (\mathsf{old}(d) + (c+2))$ (b related to $c+2$), which results in `x()`'s postcondition $d = ((c+3) + (c+2))$. We were able to prove, using ESC/Java2, that the contract of `x()` was indeed correct.

If the postcondition of `y()` is $\mathsf{result} \geq (a+2)$ instead of $\mathsf{result} = (a+2)$, the method call `y()` in `z(y(c))` cannot be related. However, we can generalise the previous solution in such a way that it gives

$$(\forall r_0 . r_0 \geq (c+2) \Rightarrow r_0 > 5) \equiv (c+2) > 5$$

as `x()`'s precondition, where r_0 is a symbolic variable denoting the result of `y()`. Unfortunately, ESC/Java2 says that the precondition of `z()` is possibly not satisfied, probably due to limitations of its theorem prover Simplify. Therefore, this generalisation is not used in our algorithm.

We have also defined propagation rules for the `return` $expr_1$ statement: we simply propagate the precondition of $expr_1$; we add to the postcondition of $expr_1$ the postcondition $\mathsf{result} = expr_1$, trying to relate any method calls in $expr_1$ to their $\mathsf{result} = expr_2$ postcondition. Essentially, a `return` statement is nothing more than a simplified assignment, to a result variable.

6.2.2 Assignments

Consider the assignment $v := expr_1$. Like ordinary wp calculus, all occurrences of the target, v, in the propagated preconditions are replaced with the RHS, $expr_1$. Essentially, the propagating postconditions part of our algorithm updates the path condition with postcondition $v = expr_1$ and integrates propagated postconditions with occurrences of v into $v = expr_1$, which is propagated; as occurrences of v may appear in the RHS, we replace them with $\mathsf{old}(v)$. As method calls may appear in the RHS, we try to replace them in $expr_1$ with $expr_2$ from their $\mathsf{result} = expr_2$ postconditions. This also makes that we need to propagate pre- and postconditions of method calls.

For example, consider assignment $x := x + y(e)$ (see Code Snippet 11) and propagated precondition $x > 1$ (x and e ints): the occurrence of x in the precondition is replaced with the adapted RHS of the assignment, $x + (e + 2)$, so the precondition is changed to $(x + (e + 2)) > 1$. Now consider the same assignment $x := x + y(e)$, but now with $x = (\mathsf{old}(x) + 1)$ as the path condition *and* the propagated postcondition: the assignment is converted into the postcondition $x = (\mathsf{old}(x) + (e + 2))$; accordingly, the path condition is updated to $x_0 = (\mathsf{old}(x)+1) \wedge x = (x_0+(e+2))$ and the propagated postcondition $x = (\mathsf{old}(x) + 1)$ is integrated into the postcondition $x = (\mathsf{old}(x) + (e + 2))$, which yields $x = ((\mathsf{old}(x) + 1) + (e + 2))$ as postcondition. The postcondition $d = (e + 3)$ of `y()` is also propagated.

6.2.3 Branching

Consider the branching `if` *expr* `then` s_1 `else` s_2. Like wp calculus, our propagating preconditions algorithm is applied to both branches, s_1 and s_2, and both outcomes are combined: *expr* and the precondition from s_1 *or* $\neg$*expr* and the precondition from s_2 should be satisfied. As for the postcondition part of our algorithm, the path condition is used to determine whether can be inferred which branch will be taken. If this is the case, our propagating postconditions algorithm is simply applied to this branch and the other branch is ignored. If not, the algorithm is applied to both branches and the outcomes are combined: the postconditions and path conditions from s_1 *or* from s_2 are valid. Before the algorithm is applied to a branch, the path condition and any propagated postconditions are extended by *expr* or its negation. Method calls may appear in *expr*; they are replaced in the same way as is done for assignments. This also makes that pre- and postconditions of method calls in *expr* need to be propagated.

For convenience, we repeat here Code Snippet 3:

Code Snippet 14 Body `branch(boolean b)`

```
if (b) beginTransaction();
else   commitTransaction();
```

Where the original algorithm outputs

$$(TRANS = 0 \wedge TRANS = 1) \equiv \mathit{false}$$

as precondition, ours outputs the precise precondition:

$$(b \wedge TRANS = 0) \vee (\neg b \wedge TRANS = 1)$$

Our propagating postconditions algorithm evaluates the method body for each precondition combination. In one of these evaluations, the path condition is initialized to $b \wedge TRANS = 0$. As this implies the condition of the `if` statement, b, the algorithm only considers the first branch. Our algorithm generates

$$(b \wedge \mathsf{old}(TRANS) = 0) \Rightarrow TRANS = 1$$
$$(\neg b \wedge \mathsf{old}(TRANS) = 1) \Rightarrow TRANS = 0$$

as postcondition for `branch()`, where the old algorithm outputs the imprecise predicate:

$$TRANS = 1 \vee TRANS = 0$$

Now suppose that the calculated precondition of `branch()` would be *true*, so the method body would be evaluated only once and the path condition would be initialized to *true*. As the path condition now would *not* imply the `if` condition, the algorithm would consider both branches and would output as postcondition:

$$\mathit{true} \Rightarrow ((b \wedge TRANS = 1) \vee (\neg b \wedge TRANS = 0))$$

For simplicity, we treat `while` loops as an `if` branch with only the `then` part; an extensive treatment would require fixed point calculations, which are outside the scope of this research, and/or user-provided loop invariants, which contradicts our intention that the user should not have to provide any annotations. Further investigation is necessary for determining whether the loop invariant generation techniques from [8] could be useful for our purposes.

6.2.4 Externalizing contract

Local variables may appear in postconditions. As long as their names are unique, e.g. not equal to instance variables, they are not a problem in the mathematical abstraction. However, a JML implementation requires a more careful treatment. Consider `ext()`, which returns an int:

Code Snippet 15 Body `ext()`

```
int a=2, b=3;
return (a+b);
```

The postcondition propagation algorithm we developed so far, gives $a = 2 \wedge b = 3 \wedge \mathsf{result} = (a + b)$. However, `a` and `b` cannot be used in the JML contract of `ext()` as they are local to `ext()`. We solve this issue by introducing a ghost variable for each local variable in the postcondition; every occurrence of this local variable in the postcondition is replaced with its ghost counterpart. In the example, the ghost variables `a0` and `b0` are introduced (declared) and the JML postcondition is changed to:

```
@ ensures a0==2 && b0==3 && \result==(a0+b0)
```

We now need to ensure that $a_0 = 2$ and $b_0 = 3$ are established by `ext()`'s body; just before the `return` statement, we set the ghost variables to their non-ghost counterparts:

```
@ set a0 = a, b0 = b
```

6.3 Comparison to old algorithm

We have seen a number of examples where our algorithm outperforms the old one. We give a summary here.

- When the second CaseSet in a method in fact adds a condition that (eventually) leads to the halted cp, our algorithm strengthens the precondition, where the old algorithm does not (Section 6.1.1).
- When the implications in the postconditions of the generated `_pre*()` and `_post*()` methods cannot be optimised away, the old algorithm gives an incorrect contract for the method `*`, where our algorithm outputs a correct contract (Section 6.1.2).
- In the complex `thrice()` example, the preconditions and postconditions of the three method calls *all* need to be integrated with each other. Our algorithm does this, resulting in a strengthened precondition and a precise postcondition; the old algorithm does *not*, resulting in a too weak, thus incorrect, precondition and an incorrect postcondition. (Section 6.1.3)
- In the branching example, the old algorithm outputs an unsatisfiable precondition and a postcondition that always evaluates to *true*, in other words a useless contract; our algorithm outputs a useful and precise contract (Section 6.2.3).

6.4 Relation with wp calculus

Pavlova *et al.* [9] defined a relation between their precondition propagation algorithm and an abstract version of wp calculus: a version that only considers static ghost variables. Our precondition propagation algorithm bears a strong resemblance to normal, non-abstract, wp calculus, e.g. our algorithm mimics the wp calculus way of evaluating a method body: from end to beginning.

We want to highlight the relation concerning sequential method calls. Consider the contract of `sendSMS()` (Code Snippet 9) with the implication optimised away, and a method that calls `sendSMS()` twice. Wp calculus [7] gives

$$\mathsf{pre}(\texttt{sendSMS}) \wedge$$
$$\forall \mathsf{mod}(\texttt{sendSMS}).\mathsf{post}(\texttt{sendSMS}) \Rightarrow \mathsf{pre}(\texttt{sendSMS})$$

as wp, resulting in:

$$(\mathsf{cp} = s_1 \wedge n < N) \wedge$$
$$\forall cp_0, n_0 . (cp_0 = s_1 \wedge n_0 = n + 1) \Rightarrow (cp_0 = s_1 \wedge n_0 < N)$$

Our precondition propagation algorithm outputs:

$$(\mathsf{cp} = s_1 \wedge n < N) \wedge (n + 1) < N \quad .$$

It turns out that the predicates are equivalent to each other, but the advantage of the latter is the absence of the universal quantification; a large number of universal quantifications are difficult to read and can be difficult for static checkers to handle. Also note that the wp that wp calculus gives contains "internal" proof obligations, e.g. $\forall cp_0.(cp_0 = s_1) \Rightarrow (cp_0 = s_1)$; our algorithm extracts the "external" part of the wp that wp calculus gives.

7. FROM PROPERTY AUTOMATA TO FULLY ANNOTATED CODE

In this section, we outline the steps that are to be taken to go from property automata to fully annotated code. The first three steps are explained in detail in [6]; the last two steps are new and explained in this paper (Sections 5-6).

- A partial PA is converted to its total one.
- The total PA is translated to method-level CaseSets.
- These CaseSets are inlined into the bodies.
- These inlined CaseSets are converted to methods with appropriate annotations and bodies; the inlined CaseSets are replaced with calls to these methods.
- These core annotations are propagated to all methods (indirectly) invoking the core-annotated ones.

The fully annotated code is given as input to the checker.

8. RELATED WORK

Aktug and Naliuka [1] developed a security policy specification language, called ConSpec, that is similar to property automata. They proved that a monitor can be inlined into the application's bytecode and then use a wp computation that essentially works the same as our propagation algorithm. In contrast, our algorithm is defined for source code and JML, which makes that we can use JML static checkers; if the checker discovers a possible violation of a security policy, it will be much more clear in what way this policy is violated if the error is given at source code level rather than at bytecode level.

Houdini [5] is a JML annotation assistant. It generates many candidate annotations, based on heuristics, e.g. it adds preconditions to methods that requires that a reference parameter is not null. It could be used in tandem with our algorithm: our algorithm *weaves* the security policy throughout the application, where Houdini adds annotations that are not security related, but need to be provided to make the *full* application verifiable.

For more related work, see the relevant sections in [6, 9].

9. CONCLUSIONS

We have developed an algorithm that overcomes the most serious limitations of the old algorithm: It supports the inlined CaseSets and propagates (core) pre- and postconditions more precisely, taking into account many concerns, such as implications in the conditions; our algorithm can perfectly handle complex (core) pre- and postconditions, where the old algorithm cannot. Essentially, our algorithm generates the correct wps and strongest postconditions in many cases, where the old algorithm generates too weak wps and incorrect postconditions too often.

We have identified a relation between the precondition part of our algorithm and wp calculus, especially a relation concerning sequential method calls; where wp calculus introduces universal quantification, our algorithm does not, which makes the preconditions simpler. They are also simpler because our algorithm extracts only the "external" proof obligations; wp calculus includes the "internal" proof obligations as well.

We have applied the algorithm to complex examples, which demonstrated that our algorithm was able to infer correct and precise method contracts, where the old algorithm failed in all but the most simple cases, resulting in many false alarms by the checker. We have used our algorithm to prove that in the 'limited SMS' example correct usages were indeed correct and to reveal a security violation.

9.1 Future work

The correctness of our conversion and propagation algorithm is not yet proved. Also, we have not yet implemented the algorithm in a tool. This tool is needed to apply our algorithm on larger examples and case studies, which would validate our method more extensively. To this end, more language constructs have to be supported: `try-catch-finally` statements and, related, exceptional postconditions.

The to be developed tool should accept (inlined) core JML-annotated code and output fully JML-annotated code. Its output is to be given as input to a checker; the results will indicate to what extent our method is able to reveal security violations in larger examples and case studies.

10. ACKNOWLEDGMENTS

I would like to thank Lodewijk Bergmans, Stefan Blom, Gijs Kant, Mariëlle Stoelinga, my peer reviewers and especially Marieke Huisman for their help, advice and support.

11. REFERENCES

[1] I. Aktug and K. Naliuka. ConSpec: A Formal Language for Policy Specification. In *REM'07*, volume 197-1 of *ENTCS*, pages 45-58, 2007.

[2] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An Overview of JML Tools and Applications. In T. Arts and W. Fokkink, editors, *FMICS'03*, volume 80 of *ENTCS*, pages 73-89, 2003.

[3] L. Burdy, A. Requet, and J.-L. Lanet. Java Applet Correctness: A Developer-Oriented Approach. In K. Araki, S. Gnesi and D. Mandrioli, editors, *FME'03*, volume 2805 of *LNCS*, pages 422-439, 2003.

[4] D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS'04*, volume 3362 of *LNCS*, pages 108-128, 2005.

[5] C. Flanagan and K.R.M. Leino. Houdini, an Annotation Assistant for ESC/Java. In J.N. Oliveira and P. Zave, editors, *FME'01*, volume 2021 of *LNCS*, pages 500-517, 2001.

[6] M. Huisman and A. Tamalet. A Formal Connection between Security Automata and JML Annotations. In M. Checkik and M. Wirsing, editors, *FASE'09*, volume 5503 of *LNCS*, pages 340-354, 2009.

[7] K.R.M. Leino. *Toward Reliable Modular Programs.* PhD thesis. Technical Report Caltech-CS-TR-95-03, California Institute of Technology, 1995.

[8] C. Păsăreanu and W. Visser. Verification of Java Programs Using Symbolic Execution and Invariant Generation. In S. Graf and L. Mounier, *SPIN'04*, volume 2989 of *LNCS*, pages 164-181, 2004.

[9] M. Pavlova, G. Barthe, L. Burdy, M. Huisman and J.-L. Lanet. 2004. Enforcing High-level Security Properties for Applets. In J.-J. Quisquater, P. Paradinas, Y. Deswarte and A.A. El Kalam, editors, *CARDIS'04*, volume 153 of *IFIP*, pages 1-16, 2004.

[10] F.B. Schneider. *Enforceable Security Policies.* Technical Report TR99-1759, Cornell University, 1999.

APPENDIX

An extended version, with the branching rules and some explanations, is available online: `http://goo.gl/bU1v9`

A. DEFINITIONS

A.1 Domains

- BinOp of binary and UnOp of unary operators
- CSE of CaseSet elements (pairs)
- CPC of control point constants
- $\mathsf{CPC_s}$ of control point constants, except halted
- Expr of expressions
- GhostAssign of assignments to ghost variables
- GhostVar of ghost variables
- MethDecl of method declarations $[name\colon \mathsf{MethName}, decls\colon \mathsf{VarDecl}^*, resType\colon \mathsf{Type}]$
- MethName of method names
- Pred of predicates
- SetStmt of JML `set` statements
- Stmt of Java statements
- Type of types (including primitive ones)
- VarDecl of variable declarations $[name\colon \mathsf{Var}, type\colon \mathsf{Type}]$
- Var of variables (names)

A.2 Auxiliary functions

- $\mathsf{body}\colon \mathsf{MethName} \to \mathsf{Stmt}$; denoting method body
- call; denoting method call (with $\geq$ zero arguments)
- $\mathsf{curr}\colon \mathsf{Pred} \to \mathsf{Pred}$; returns the predicate with all pre-invocation values replaced with their current values
- $\mathsf{fv}\colon \mathsf{Pred} \to \mathcal{P}(\mathsf{Var})$; returns the set of free variables
- $\mathsf{lVars}\colon \mathsf{MethName} \to \mathcal{P}(\mathsf{Var})$; returns the names of all local variables declared in the given method
- methDecl; returns method declaration
- mod; returns all assigned variables in the given statement/expression/method (see online for definition)
- modP, notModP; returns those of the given predicates in which the given variables do not appear (see online for formal definition)
- $\mathsf{old}\colon \mathsf{Pred} \to \mathsf{Pred}$; returns the given predicate with all variables replaced with their pre-invocation values
- $\mathsf{recent}\colon \mathsf{Var} \to \mathsf{Pred} \to \mathsf{Var}$; returns the highest of the (un)numbered versions of the variable in the given predicate, e.g. $\mathsf{recent}(n, n_0 = (n + n_1)) = n_1$ and $\mathsf{recent}(n, n = 2) = n$ *(if no (un)numbered version of the variable exists in the predicate, its pre-invocation value is returned, e.g.* $\mathsf{recent}(n, x = 2) = \mathsf{old}(n)$*)*
- $\mathsf{subp}\colon \mathsf{Pred} \to \mathcal{P}(\mathsf{Pred})$; returns the set of all (sub) predicates of the given predicate
- $\mathsf{type}\colon \mathsf{Expr} \to \mathsf{Type}$; returns type of given expression
- $\mathsf{unique}\colon \mathsf{Var} \to \mathsf{Pred} \to \mathsf{Var}$; counterpart of recent; returns lowest of the numbered versions of the given variable name that do not (currently) exist in the given predicate and do not exist as ghost variables (in the class) either, e.g. $\mathsf{unique}(n, n_0 = (n + n_1)) = n_2$

A.3 Notation

We will use the subscripts *conj* and *disj* to take a predicate as the set of conjuncts and disjuncts, so $(p = 1 \wedge q)_{conj} = \{p = 1, q\}$ and $(p = 1 \vee q)_{disj} = \{p = 1, q\}$. We introduce the boolean $\sim$ operator which returns *true* iff a predicate (first operand) has a particular format (second operand), so $(c = 1 \Rightarrow (d = 2 \wedge e)) \sim [a \Rightarrow b]$ returns *true*, while $(c = 1) \sim [a \Rightarrow b]$ returns *false*. We use $\perp$ to denote the outcome of an expression whose evaluation is undefined. We denote Hilbert's choice operator with ε.

B. CONVERSION ALGORITHM

For every method $[m\colon \mathsf{MethName}, decls\colon \mathsf{VarDecl}^*$ and $res\colon \mathsf{Expr}]$ that has (two) CaseSets:

rewrite $[\mathsf{CaseSet}\ E_1; body; \mathsf{CaseSet}\ E_2]$ to:
$[\mathsf{call}(m_1, decls_{names}); body; \mathsf{call}(m_2, res)]$
where $E_1, E_2\colon \mathcal{P}(\mathsf{CSE})$, $body\colon \mathsf{Stmt}$ *and* $m_1, m_2\colon$ MethName *(new) names of two new methods:*

- $\mathbf{methDecl}(m_1) = [m_1, decls, \mathsf{void}]$, $\mathbf{pre}(m_1) = \mathsf{csPre}(E_1)$, $\mathbf{mod}(m_1) = \mathsf{csMod}(E_1)$, $\mathbf{post}(m_1) = \mathsf{csPost}(E_1)$, $\mathbf{body}(m_1) = \mathsf{csSetAutVars}(E_1); \mathsf{csSetCP}(E_1)$
- $\mathbf{methDecl}(m_2) = [m_2, [[\mathsf{result}, \mathsf{type}(res)]], \mathsf{void}]$, $\mathbf{pre}(E_2) = \mathsf{csPre}(E_2)$, $\mathbf{mod}(m_2) = \mathsf{csMod}(E_2)$, $\mathbf{post}(m_2) = \mathsf{csPost}(E_2)$, $\mathbf{body}(m_2) = \mathsf{csSetAutVars}(E_2); \mathsf{csSetCP}(E_2)$

($s, s_1, s_2, s_3\colon \mathsf{CPC}$, $A\colon \mathcal{P}(\mathsf{GhostAssign})$ *and* $g\colon \mathsf{Pred}$*):*

- $\mathsf{csPre}\colon \mathcal{P}(\mathsf{CSE}) \to \mathsf{Pred}$
  $\mathbf{csPre}(E) = \bigvee\{\mathsf{cp} = s_1 \wedge g \mid [\mathsf{cp} = s_1 \wedge g, \mathsf{cp} := s_2; A] \in E \wedge s_2 \neq \mathsf{halted}\}$
- $\mathsf{csMod}\colon \mathcal{P}(\mathsf{CSE}) \to \mathcal{P}(\mathsf{GhostVar})$
  $\mathbf{csMod}(E) = \bigcup_{[\mathsf{cp}=s\wedge g,\ A]\in E}\{n \mid [n := expr] \in A\}$
- $\mathsf{csPost}\colon \mathcal{P}(\mathsf{CSE}) \to \mathcal{P}(\mathsf{Pred})$
  $\mathbf{csPost}(E) = \mathsf{\_csPost}_1(E) \cup \{\mathsf{\_csPost}_2(E)\}$
- $\mathsf{\_csPost}_1\colon \mathcal{P}(\mathsf{CSE}) \to \mathcal{P}(\mathsf{Pred})$
  $\mathbf{\_csPost}_1(E) = \bigcup_{[\mathsf{cp}=s_1\wedge g,\ \mathsf{cp}:=s_2;A]\in E\wedge s_2\neq\mathsf{halted}}$
  $\mathsf{old}(\mathsf{cp} = s_1 \wedge g) \Rightarrow \bigwedge(\mathsf{cp} = s_2, \{n = \mathsf{old}(expr_1) \mid [n_1 := expr_1] \in A\}, \{n_2 = \mathsf{old}(n_2) \mid n_2 \in \mathsf{csMod}(E)\backslash\{\mathsf{cp}\} \wedge \neg\exists[n_3 := expr_2]\colon A.n_2 = n_3\})$
- $\mathsf{\_csPost}_2\colon \mathcal{P}(\mathsf{CSE}) \to \mathsf{Pred}$
  $\mathbf{\_csPost}_2(E) = false \Rightarrow \bigwedge\{\mathsf{cp} = s_1 \mid s_1 \in \mathsf{cpc_s} \wedge \neg\exists[\mathsf{cp} = s_2 \wedge g, \mathsf{cp} := s_3; A]\colon E.s_1 = s_3\}$
- $\mathsf{csSetAutVars}\colon \mathcal{P}(\mathsf{CSE}) \to \mathcal{P}(\mathsf{SetStmt})$
  $\mathbf{csSetAutVars}(E) = \bigcup_{[\mathsf{cp}=s_1\wedge g,\ \mathsf{cp}:=s_2;A]\in E\wedge s_2\neq\mathsf{halted}}$
  $\{\mathtt{set}\ n := (\mathsf{cp} = s_1 \wedge g)\ ?\ expr : n \mid [n := expr] \in A\}$
- $\mathsf{csSetCP}\colon \mathcal{P}(\mathsf{CSE}) \to \mathsf{SetStmt}$
  $\mathbf{csSetCP}(E) = \mathtt{set}\ \mathsf{cp} := \mathsf{\_csCPExpr}(E)$
- $\mathsf{\_csCPExpr}\colon \mathcal{P}(\mathsf{CSE}) \to \mathsf{Expr}$
  $\mathbf{\_csCPExpr}(\emptyset) = \mathsf{cp}$
  $\mathbf{\_csCPExpr}(\{[\mathsf{cp} = s_1 \wedge g, \mathsf{cp} := s_2; A], ...\}) =$

$$\begin{cases} \mathsf{\_csCPExpr}(...) & \textbf{if } s_2 = \mathsf{halted} \\ (\mathsf{cp} = s_1 \wedge g)\ ?\ s_2 : \mathsf{\_csCPExpr}(...) & \textbf{if } s_2 \neq \mathsf{halted} \end{cases}$$

C. PROPAGATION ALGORITHM

($exprs\colon \mathsf{Expr}^*$ *and* $n\colon$ MethName*):*

- $\mathsf{expr}\colon \mathsf{Expr} \to \mathsf{Expr}$
  $\mathbf{expr}(expr) = expr[\mathsf{\_expr}(\mathsf{call}(n, exprs))/\mathsf{call}(n, exprs)$ **for each** $\mathsf{call}(n, exprs)$ **in** $expr]$
- $\mathsf{\_expr}\colon \mathsf{Expr} \to \mathsf{Expr}$
  $\mathbf{\_expr}(\mathsf{call}(n, exprs)) =$

$$\begin{cases} \mathsf{curr}(expr) & \textbf{if } \exists!p\colon \mathsf{post}(n, exprs).p \sim [\mathsf{result} = expr] \\ \perp & \textbf{otherwise} \end{cases}$$

  (where $[\mathsf{result} = expr] = \varepsilon\, p\colon \mathsf{post}(n, exprs).p \sim [\mathsf{result} = expr]$***)***
- $\mathsf{rnm}\colon \mathcal{P}(\mathsf{Pred}) \to \mathsf{Expr}^* \to \mathsf{VarDecl}^* \to \mathcal{P}(\mathsf{Pred})$
  $\mathbf{rnm}(P, exprs, decls) = \{p[\mathsf{expr}(expr)/name$ **for each** $expr$ **in** $exprs$ ***(where*** $[name\colon \mathsf{Var}, type\colon \mathsf{Type}]$ ***of corresponding variable declaration in*** $decls$***)*** $] \mid p \in P\}$

C.1 Propagation of preconditions

C.1.1 Sequential method calls

We define functions $\mathbf{pre}\colon \mathsf{MethName} \to \mathcal{P}(\mathsf{Pred})$ and $\mathbf{pre}\colon \mathsf{Stmt} \to \mathcal{P}(\mathsf{Pred}) \to \mathcal{P}(\mathsf{Pred})$ by rules like *(*$m, n :$ MethName*,* $PRE : \mathcal{P}(\mathsf{Pred})$ *and* $s_1, s_2 :$ Stmt*):*

$\mathbf{pre}(m) = \mathsf{pre}(\mathsf{body}(m), \emptyset)$
$\mathbf{pre}(s_1;s_2, PRE) = \mathsf{pre}(s_1, \mathsf{pre}(s_2, PRE))$
$\mathbf{pre}(\mathsf{call}(n), PRE) = \mathsf{pre}(n) \cup \mathsf{notModP}(PRE, \mathsf{mod}(n)) \cup \bigcup_{pre\in\mathsf{modP}(PRE,\mathsf{mod}(n))} \mathsf{prsrv}(pre, \mathsf{post}(n), \mathsf{mod}(n))$

- $\mathsf{prsrv}\colon \mathsf{Pred} \to \mathcal{P}(\mathsf{Pred}) \to \mathcal{P}(\mathsf{Var}) \to \mathcal{P}(\mathsf{Pred})$
  $\mathbf{prsrv}(pre, POST, V) =$
  $$\begin{cases} \_\mathsf{prsrv}(pre, POST) & \textbf{if all occurrences of } V \textbf{ in } pre \\ & \quad \textbf{can be transformed by } POST \\ \emptyset & \textbf{otherwise} \end{cases}$$
- $\_\mathsf{prsrv}\colon \mathsf{Pred} \to \mathcal{P}(\mathsf{Pred}) \to \mathcal{P}(\mathsf{Pred})$
  $\mathbf{\_prsrv}(pre, POST) =$
  $\mathsf{prsrv}_3(\mathsf{prsrv}_2(\mathsf{prsrv}_1(pre, POST), POST), POST)$
- $\mathsf{prsrv}_1\colon \mathsf{Pred} \to \mathcal{P}(\mathsf{Pred}) \to \mathsf{Pred}$
  $\mathbf{prsrv}_1(pre, POST) = pre[\{\bigvee A/subp \mid subp \in \mathsf{subp}(pre) \wedge A = \mathsf{antcdnts}(POST, subp) \wedge A \neq \emptyset\}]$
- $\mathsf{antcdnts}\colon \mathcal{P}(\mathsf{Pred}) \to \mathsf{Pred} \to \mathcal{P}(\mathsf{Pred})$
  $\mathbf{antcdnts}(POST, conseq) = \{\mathsf{curr}(a) \mid post \in POST \wedge (post \sim [a \Rightarrow b] \wedge conseq \in b_{conj})\}$
- $\mathsf{prsrv}_2\colon \mathsf{Pred} \to \mathcal{P}(\mathsf{Pred}) \to \mathsf{Pred}$
  $\mathbf{prsrv}_2(pre, POST) =$
  $pre[\bigcup_{post \in POST \wedge post \sim [a=b]} \{\mathsf{curr}(b)/a, \mathsf{curr}(a)/b\}]$
- $\mathsf{prsrv}_3\colon \mathsf{Pred} \to \mathcal{P}(\mathsf{Pred}) \to \mathcal{P}(\mathsf{Pred})$
  $\mathbf{prsrv}_3(pre, POST) =$
  $$\begin{cases} \{pre\} & \textbf{if } \_\mathsf{prsrv}_3(pre, POST) = \emptyset \\ \_\mathsf{prsrv}_3(pre, POST) & \textbf{if } \_\mathsf{prsrv}_3(pre, POST) \neq \emptyset \end{cases}$$
- $\_\mathsf{prsrv}_3\colon \mathsf{Pred} \to \mathcal{P}(\mathsf{Pred}) \to \mathcal{P}(\mathsf{Pred})$
  $\mathbf{\_prsrv}_3(pre, POST) = \{\mathsf{curr}(a) \Rightarrow \_\mathsf{prsrv}_3(pre, conseq) \mid post \in POST \wedge (post \sim [a \Rightarrow conseq] \wedge \_\mathsf{prsrv}_3(pre, conseq) \neq pre)\}$
- $\_\mathsf{prsrv}_3\colon \mathsf{Pred} \to \mathsf{Pred} \to \mathsf{Pred}$
  $\mathbf{\_prsrv}_3(pre, conseq) =$
  $pre[\bigcup_{p \in conseq_{conj} \wedge p \sim [a=b]} \{\mathsf{curr}(b)/a, \mathsf{curr}(a)/b\}]$

C.1.2 More language constructs

Method arguments and results.

The rule $\mathbf{pre}(\mathsf{call}(n), PRE)$ from C.1.1 is deprecated and replaced with *(*$exprs\colon \mathsf{Expr}^*$, $n\colon \mathsf{MethName}$ *and* $PRE\colon \mathcal{P}(\mathsf{Pred})$*)*:
$\mathbf{pre}(\mathsf{call}(n, exprs), PRE) = \mathsf{pre}(exprs, \_\mathsf{pre}(\mathsf{call}(n, exprs), PRE))$

We also define functions $\mathbf{\_pre}\colon \mathsf{Stmt} \to \mathcal{P}(\mathsf{Pred}) \to \mathcal{P}(\mathsf{Pred})$, $\mathbf{pre}\colon \mathsf{MethName} \to \mathsf{Expr}^* \to \mathcal{P}(\mathsf{Pred})$, $\mathbf{pre}\colon \mathsf{Expr}^* \to \mathcal{P}(\mathsf{Pred}) \to \mathcal{P}(\mathsf{Pred})$ and $\mathbf{pre}\colon \mathsf{Expr} \to \mathcal{P}(\mathsf{Pred}) \to \mathcal{P}(\mathsf{Pred})$ by rules like *(*$bop\colon \mathsf{BinOp}$, $expr, expr_1, expr_2\colon \mathsf{Expr}$, $exprs\colon \mathsf{Expr}^*$, $n\colon \mathsf{MethName}$, $PRE\colon \mathcal{P}(\mathsf{Pred})$ *and* $uop\colon \mathsf{UnOp}$*)*:

$\mathbf{\_pre}(\mathsf{call}(n, exprs), PRE) = \mathsf{pre}(n, exprs) \cup \mathsf{notModP}(PRE, \mathsf{mod}(n)) \cup \bigcup_{pre \in \mathsf{modP}(PRE, \mathsf{mod}(n))} \mathsf{prsrv}(pre, \mathsf{post}(n, exprs), \mathsf{mod}(n))$
$\mathbf{pre}(n, exprs) = \mathsf{rnm}(\mathsf{pre}(n), exprs, decls)$ ***(where*** $[name\colon \mathsf{MethName}, decls\colon \mathsf{VarDecl}^*, resType\colon \mathsf{Type}] = \mathsf{methDecl}(n)$***)***
$\mathbf{pre}([expr_1^*, expr_2], PRE) = \mathsf{pre}(expr_1^*, \mathsf{pre}(expr_2, PRE))$
$\mathbf{pre}(expr_1\ bop\ expr_2, PRE) = \mathsf{pre}(expr_1, \mathsf{pre}(expr_2, PRE))$
$\mathbf{pre}(uop\ expr, PRE) = \mathsf{pre}(expr, PRE)$

We add to $\mathbf{pre}\colon \mathsf{Stmt} \to \mathcal{P}(\mathsf{Pred}) \to \mathcal{P}(\mathsf{Pred})$ the rule *(*$expr\colon \mathsf{Expr}$*)*: $\mathbf{pre}(\texttt{return}\ expr, PRE) = \mathsf{pre}(expr, PRE)$

Assignments.

We add to $\mathbf{pre}\colon \mathsf{Stmt} \to \mathcal{P}(\mathsf{Pred}) \to \mathcal{P}(\mathsf{Pred})$ the rule *(*$expr\colon \mathsf{Expr}$ *and* $v\colon Var$*)*:
$\mathbf{pre}(v := expr, PRE) =$
$\mathsf{pre}(expr, \{pre[\mathsf{expr}(expr)/v] \mid pre \in PRE\})$

C.2 Propagation of postconditions

C.2.1 Sequential method calls

We define functions $\mathbf{post}\colon \mathsf{MethName} \to \mathcal{P}(\mathsf{Pred})$ and $\mathbf{post}\colon \mathsf{Stmt} \to [\mathsf{Pred}, \mathcal{P}(\mathsf{Pred})] \to [\mathsf{Pred}, \mathcal{P}(\mathsf{Pred})]$ by rules *(*$m, n : \mathsf{MethName}$, $path\colon \mathsf{Pred}$, $POST : \mathcal{P}(\mathsf{Pred})$ *and* $s_1, s_2 : \mathsf{Stmt}$*)*:

$\mathbf{post}(m) = \{\mathsf{old}(\bigwedge pre') \Rightarrow \bigwedge POST \mid pre' \in \times\{pre_{disj} \mid pre \in \mathsf{pre}(m)\} \wedge [path, POST] = \mathsf{post}(\mathsf{body}(m), [\bigwedge pre', \emptyset])\}$
$\mathbf{post}(s_1;s_2, [path, POST]) = \mathsf{post}(s_2, \mathsf{post}(s_1, [path, POST]))$
$\mathbf{post}(\mathsf{call}(n), [path, POST]) =$
$[\mathsf{updPath}(path, \mathsf{applicable}(path, \mathsf{post}(n)), \mathsf{mod}(n)), \mathsf{notModP}(POST, \mathsf{mod}(n)) \cup \bigcup_{post \in \mathsf{applicable}(path, \mathsf{post}(n))} \mathsf{integrate}(\mathsf{modP}(POST, \mathsf{mod}(n)), post)]$

- $\mathsf{applicable}\colon \mathsf{Pred} \to \mathcal{P}(\mathsf{Pred}) \to \mathcal{P}(\mathsf{Pred})$
  $\mathbf{applicable}(path, POST) = \{b \mid post_1 \in POST \wedge (post_1 \sim [a \Rightarrow b] \wedge path \Rightarrow \mathsf{curr}(a))\} \cup \{post_2 \mid post_2 \in POST \wedge post_2 \not\sim [a \Rightarrow b]\}$
- $\mathsf{updPath}\colon \mathsf{Pred} \to \mathcal{P}(\mathsf{Pred}) \to \mathcal{P}(\mathsf{Var}) \to \mathsf{Pred}$
  $\mathbf{updPath}(path, POST, V) =$
  $\_\mathsf{updPath}(path[\{\mathsf{unique}(v, path)/v \mid v \in V\}], POST)$
- $\_\mathsf{updPath}\colon \mathsf{Pred} \to \mathcal{P}(\mathsf{Pred}) \to \mathsf{Pred}$
  $\mathbf{\_updPath}(path, POST) = path \wedge \bigwedge\{\_\mathsf{updPath}(path, post) \mid post \in POST\}$
- $\_\mathsf{updPath}\colon \mathsf{Pred} \to \mathsf{Pred} \to \mathsf{Pred}$
  $\mathbf{\_\_updPath}(path, post) =$
  $post[\{\mathsf{recent}(v, path)/\mathsf{old}(v) \mid v \in \mathsf{fv}(post)\}]$
- $\mathsf{integrate}\colon \mathcal{P}(\mathsf{Pred}) \to \mathsf{Pred} \to \mathsf{Pred}$
  $\mathbf{integrate}(PROPPOST, mPost) = mPost [\bigcup_{propPost \in PROPPOST \wedge propPost \sim [a=b]} \{b/\mathsf{old}(a), a/\mathsf{old}(b)\}]$

C.2.2 More language constructs

Method arguments and results.

The rule $\mathbf{post}(\mathsf{call}(n), [path, POST])$ from C.2.1 is deprecated and replaced with *(*$exprs\colon \mathsf{Expr}^*$, $n\colon \mathsf{MethName}$, $path\colon \mathsf{Pred}$ *and* $POST\colon \mathcal{P}(\mathsf{Pred})$*)*:

$\mathbf{post}(\mathsf{call}(n, exprs), [path, POST]) = \_\mathsf{post}(\mathsf{call}(n, exprs), \mathsf{post}(exprs, [path, POST]))$

We also define functions $\mathbf{\_post}\colon \mathsf{Stmt} \to [\mathsf{Pred}, \mathcal{P}(\mathsf{Pred})] \to [\mathsf{Pred}, \mathcal{P}(\mathsf{Pred})]$, $\mathbf{post}, \mathbf{postf}\colon \mathsf{MethName} \to \mathsf{Expr}^* \to \mathcal{P}(\mathsf{Pred})$, $\mathbf{post}\colon \mathsf{Expr}^* \to [\mathsf{Pred}, \mathcal{P}(\mathsf{Pred})] \to [\mathsf{Pred}, \mathcal{P}(\mathsf{Pred})]$ and $\mathbf{post}\colon \mathsf{Expr} \to [\mathsf{Pred}, \mathcal{P}(\mathsf{Pred})] \to [\mathsf{Pred}, \mathcal{P}(\mathsf{Pred})]$ by rules *(*$bop\colon \mathsf{BinOp}$, $expr, expr_1, expr_2\colon \mathsf{Expr}$, $exprs\colon \mathsf{Expr}^*$, $n\colon \mathsf{MethName}$, $path\colon \mathsf{Pred}$, $POST\colon \mathcal{P}(\mathsf{Pred})$ *and* $uop\colon \mathsf{UnOp}$ *)*:

$\mathbf{\_post}(\mathsf{call}(n, exprs), [path, POST]) = [\mathsf{updPath}(path, \mathsf{applicable}(path, \mathsf{postf}(n, exprs)), \mathsf{mod}(n)), \mathsf{notModP}(POST, \mathsf{mod}(n)) \cup \bigcup_{post \in \mathsf{applicable}(path, \mathsf{postf}(n, exprs))} \mathsf{integrate}(\mathsf{modP}(POST, \mathsf{mod}(n)), post)]$
$\mathbf{postf}(n, exprs) = \{p \mid p \in \mathsf{post}(n, \mathsf{exprs}) \wedge p \not\sim [\mathsf{result} = expr]\}$
$\mathbf{post}(n, exprs) = \mathsf{rnm}(\mathsf{post}(n), exprs, decls)$ ***(where*** $[name\colon \mathsf{MethName}, decls\colon \mathsf{VarDecl}^*, resType\colon \mathsf{Type}] = \mathsf{methDecl}(n)$***)***
$\mathbf{post}([expr_1, expr_2^*], [path, POST]) = \mathsf{post}(expr_2^*, \mathsf{post}(expr_1, [path, POST]))$
$\mathbf{post}(expr_1\ bop\ expr_2, [path, POST]) = \mathsf{post}(expr_2, \mathsf{post}(expr_1, [path, POST]))$
$\mathbf{post}(uop\ expr, [path, POST]) = \mathsf{post}(expr, [path, POST])$

We add to $\mathbf{post}\colon \mathsf{Stmt} \to [\mathsf{Pred}, \mathcal{P}(\mathsf{Pred})] \to [\mathsf{Pred}, \mathcal{P}(\mathsf{Pred})]$ the rule *(*$expr\colon \mathsf{Expr}$*)*: $\mathbf{post}(\texttt{return}\ expr, [path_1, POST_1]) = [path_2, POST_2 \cup \{\mathsf{result} = \mathsf{expr}(expr)\}]$ ***(where*** $[path_2, POST_2] = \mathsf{post}(expr, [path_1, POST_1])$***)***

Assignments.

We add to $\mathbf{post}\colon \mathsf{Stmt} \to [\mathsf{Pred}, \mathcal{P}(\mathsf{Pred})] \to [\mathsf{Pred}, \mathcal{P}(\mathsf{Pred})]$ the rule *(*$expr\colon \mathsf{Expr}$ *and* $v\colon \mathsf{Var}$*)*: $\mathbf{post}(v := expr, [path, POST]) = \_\mathsf{post}(v := expr, \mathsf{post}(expr, [path, POST]))$

We add to $\mathbf{\_post}\colon \mathsf{Stmt} \to [\mathsf{Pred}, \mathcal{P}(\mathsf{Pred})] \to [\mathsf{Pred}, \mathcal{P}(\mathsf{Pred})]$ the rule *(*$expr\colon \mathsf{Expr}$ *and* $v\colon \mathsf{Var}$*)*: $\mathbf{\_post}(v := expr, [path, POST]) = [\mathsf{updPath}(path, v = expr[\mathsf{old}(v)/v], \{v\}), \mathsf{notModP}(POST, \{v\}) \cup \mathsf{integrate}(\mathsf{modP}(POST, \{v\}), v = expr[\mathsf{old}(v)/v])]$

Externalizing contract.

The rule $\mathbf{post}(m)$ from C.2.1 is renamed to $\mathbf{\_post}(m)$ and the rule itself is replaced with: $\mathbf{post}(m) = \_\mathsf{post}(m)[\{\mathsf{unique}(l, \bigwedge \_\mathsf{post}(m))/l \mid l \in \mathsf{lVars}(m)\}]$